SAND9x-xxxx • UC-xxx
Unlimited Release
Printed July, 1997

A DRAFT SPECIFICATION FOR THE FINITE-ELEMENT–SOLVER INTERFACE
Version 0.07

Robert L. Clay
Kyran D. Mish[¤]
Alan B. Williams
Sandia National Laboratories, Livermore, California

Lee M. Taylor
Sandia National Laboratories, Albequerque, New Mexico

Ivan Otero
Lawrence Livermore National Laboratories, Livermore, California

Abstract

This document introduces a draft specification for a solver abstraction designed for use with finite-element applications. The solution services outlined here are implemented using a layered computational architecture that is intended to simplify the task of adding equation solving services to existing and new finite-element analysis codes. The basic motivation for introducing this solver abstraction is to simplify the task of developing and supporting large-scale finite-element analysis software, and especially parallel applications.

The solution services are outlined in brief, and then successively refined until each aspect of their architecture (both algorithms and data) is made concrete.

The solver interface presented is intended to provide sufficient flexibility so that it can be used in a wide variety of finite-element applications in science and engineering. In addition, it is designed to hide the details of the equation solution process so that new hardware and software technology can be accommodated without disturbing the parent finite-element analysis program. Finally, it is intended to be sufficiently easy to use so that its installation within a finite-element application will be relatively painless.

[¤] Computational Mechanics Project, Dept. of Mechanical Engineering, California State University Chico.

Caveats

This document accompanies version 0.07 of the header file detailing the proposed interface between a finite-element (FE) program and its embedded linear algebra (LA) solver. It is a draft-quality description of the high-level concepts underlying this interface, and hence subject to change based on input from the FE developer community.

There is little detail in this current document, as the decision has consciously been made to avoid substantial detail until the specification is finalized – upon adoption of the interface as a (relatively) static entity, more examples and accompanying documentation will be provided.

In particular, all of the pictures used to diagram concepts inherent in the proposed specification are simplified by representation in only two space dimensions. This choice is merely to make the illustration process easier, and there are *no* limitations within the proposed finite-element interface that limit its utility in the three-dimensional case (in fact, the solver interface is independent of the spatial dimension of the problem being solved). The process of generalization to 3D is left to the reader, as least in the current draft stage of this document.

The current version of the interface is specifically oriented towards sparse matrix methods in general, and Krylov iterative solvers in particular. The interface is intended to be extensible to other linear and nonlinear solution algorithms in the future, but they are not the guiding principles of the present implementation.

Finally, the order in which the material is presented represents a successively more refined view of the problem of providing FE equation solution support. This leads to the problem that detailed definitions of terms arise later rather than sooner in this document, so that the important concepts of motivation and calling architecture are given only in a general sense.

Motivation

There are many motivations for the development of the current FE-LA interface, and most are based on the desire to provide a useful abstraction of the wide variety of solver services required by current and future finite-element codes. The following enumeration highlights some of the most important motivating principles.

Simplicity

Solving sparse finite-element equations (and especially in a parallel setting) is a complicated process that rests on a small set of relatively simple concepts. Element matrices must be formed and assembled into a particular sparse matrix format, boundary condition data must be embedded into the resulting partially-constructed solution, various types of constraint relations must be appended to the system, and the resulting system must be solved using a particular algorithm from a bewildering variety of possible numerical schemes for solving sparse linear systems.

The basic idea behind the interface is to separate the physics (represented by the FE calculations) from the algebra (represented by the processes mentioned above), in order to simplify code development for FE analysis. Tasks such as assembly, boundary-condition modification, implementation of linear constraints, and solution services are encapsulated into a *linear system of equations* object, and this packaging of data and services permits the FE developer to concentrate on the physics instead of on the linear algebra.

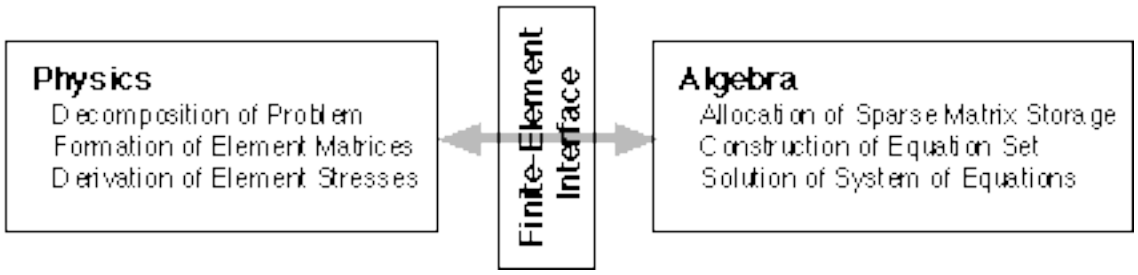


Figure: Division of Labor With the Proposed Interface

Furthermore, the proposed interface attempts to perform the algebraic computations in a sequence that reflects a typical FE analysis architecture. Data that is requested from the FE code is passed in a form that naturally occurs in most high-performance FE programs: for example, lists of element matrices are requested, with controls provided to reflect typical dense (or packed) element matrix storage schemes used in various FE programs. With this approach, the proposed interface requests data when the FE program is likely to have that data available, and permits it to be passed in a format that requires minimal (if any) transformations of the data.

Hence the guiding principles of the interface are to request data when and how it is readily available, to package services in a form to simplify the tasks of equation solution (and in fact, to remove the whole issue of sparse matrix solution from the realm of finite-element development), and to provide the "answers" in a form amenable for subsequent use by the finite-element code.

Generality

Finite-element codes are used to solve an incredible variety of problems, and this diversity of application often manifests itself in a diversity of software architectures. Some finite-element codes are designed for pure speed, so that many architectural decisions (such as the local order of the finite-element interpolant) are collectively standardized over the entire problem domain. Other codes (most notably, p-adaptive analyses, where the local element interpolation order often varies across element boundaries) reflect a more individualized elemental approach to calculation.

The interface attempts to handle either extreme of architecture by introducing abstractions for handling collections of elements and nodes. These abstract aggregations can be utilized to encompass large groups of nodes and elements, or to

degenerate gracefully to the case where data is passed one element at a time, or one node at a time.

Furthermore, the interface is designed for parallel SPMD implementation, but the parallel model does not preclude its efficient use in a serial (or shared-memory parallel) computational setting. In particular, abstractions that arise from distributed-memory parallel concerns (such as external nodes) can readily be ignored in the serial setting, and the interface implementation will still provide substantial value to the finite-element developer.

Extensibility

Just as finite-element calculations reflect a variety of computational architectures, they also provide a diverse set of *needs* for solution services. The current implementation has been designed not only to provide the lowest-common denominator of useful solution function (namely, scalable sparse linear algebraic solution services), but also to provide for future extensions common to current finite-element analyses. Some of the future extensions include:

- nonlinear solution services, which will require a tighter coupling between the physics and the algebra, in that the finite-element code will have to export services to the solver (e.g., evaluation of element residuals),
- eigenvalue computation, which will require the extension of the interface to accommodate passing element mass matrices (and of course, some new equation-solution control parameters) to the solver, and
- multilevel solution methods, some of which can be grafted onto the interface architecture in a straightforward fashion.

Hence, the proposed interface should be viewed as currently providing needed baseline function that can be generalized as more extensive needs arise.

Efficiency

Certainly, one of the most important motivations for providing a FE-LA interface is that of efficiency. As new solution algorithms are uncovered, or as new computational architectures are developed, separating the physics from the algebra permits greater efficiencies by hiding the details of the solution services so they can be made more efficient without disrupting the architecture of the parent finite-element code. Thus, one of the most important motivations behind the proposed interface is the desire to provide efficient solution algorithmic implementations that can be tuned to specific FE problems, or to a given computer architecture.

Another important efficiency concern arises from the use of adaptive codes, which require that initialization data (such as the sparse matrix realization used to assemble and solve the finite-element equation) changes often. The use of adaptive schemes (or at least h-adaptive techniques, where the order of the element interpolants doesn't change within a group of elements) for solving complicated problems leads to the need for lightweight initialization routines, and this need is reflected in the current interface specification and its prototypical implementation.

Finally, large finite-element analyses require considerable memory to run, and it is highly desirable that the equation solution modules minimize their memory footprint. The specification is designed (e.g., by separating initialization from loading stages) to provide for optimal memory allocation in its production implementation.

Overall Interface Architecture

The current interface involves four separate steps that are performed in the order given below:

Initialization

This step is used to advise the solver implementation on the overall structure of the finite-element equations, so that sparse matrix storage can be allocated, future data-passing needs estimated, and verification information stored for subsequent checking. Note that the initialization step only provides the structure of the matrix, so that particular matrix entries cannot be determined at the end of this initial task.

Loading

This step involves passing the particular data used to fill the data structures constructed during the initialization process. The data to be passed here includes such substantial demands as lists of element matrices and tables of boundary-condition data. At the end of the loading step, the values of each nonzero term in the sparse matrix are completely specified.

Solution

In this task, the resulting system of linear equations is solved in a scalable and robust manner, and in accordance with control data passed by the parent finite-element analysis program.

Solution Return

Here, the various components of the solution are exported to the finite-element program, in formats most natural for subsequent use in finite-element analysis.

Definitions

The following definitions describe the abstractions used in the interface specification. It is important to note that these concepts represent collections of finite-element data structures, so that data can be passed either in terms of standardized groups, or on an element-by-element/node-by-node approach. It is also important to note that all constructs related to parallel computation (namely shared and external nodes) can be ignored in the serial case (i.e., there are zero external or shared nodes in this case).

Fundamental Constructs

Finite-element analyses involve the construction of an interpolant that approximates the solution of a physical problem. The finite-element interpolant is constructed locally over subdomains (termed elements), and the local interpolants are implemented using a collection of interpolation nodes that are located on the

boundary of the element, or within its interior. The overall collection of nodes and elements used in a finite-element analysis is termed the *finite-element mesh*.

There is no requirement that only a single interpolant be associated with a set of elements, as finite-element models are commonly used to solve multiple-field problems. For example, incompressible flow problems often require construction of a nodal velocity field that interpolates the velocity over the problem domain in a continuous (but not necessarily smooth) manner, and also the construction of a pressure field that is only piecewise-continuous over each element. The former field is generally associated with nodes lying in or on elements, while the latter may be associated with nodes lying entirely within elements, or with purely elemental solution unknowns.

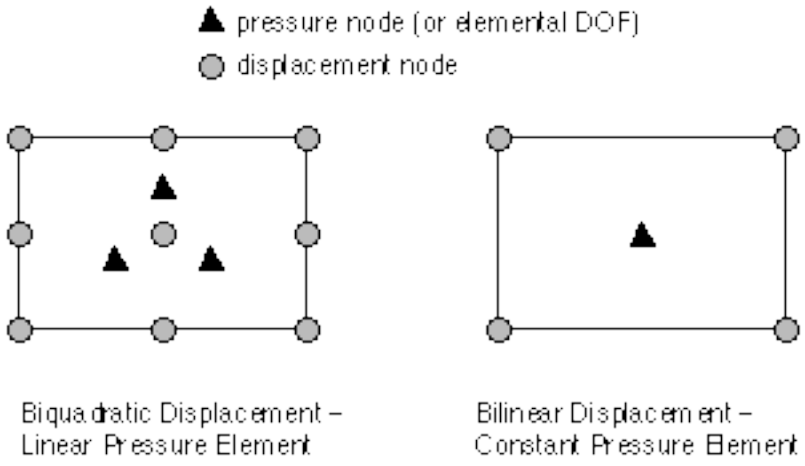


Figure: typical field variables for incompressible flow

In addition to the nodes and elements that define the particular finite-element interpolant, many finite-element codes permit various forms of constraints to be enforced between different components of the mesh. A typical example is in contact-impact calculations, where different mechanical components interact without occupying the same space at the same time. The constraint equations that arise from a precise mathematical representation of "only one object can occupy a given point at a given time" must be implemented within the finite-element equation set in order for the solution to make physical sense.

Domain Decomposition

In a parallel computing environment, the problem domain is divided into subdomains which are then associated with individual processors. This subdivision process is termed domain decomposition, and in the setting of finite-element models, corresponds to assigning groups of elements to various processors. The overall process is diagrammed in the figure below. Note how the subdivision process results in each element being mapped to a distinct processor, but that nodes lying on the boundaries of some elements may end up being associated with more than one processor. Such nodes that are associated with more than one processor are termed *shared nodes*. Nodes that are associated with only one processor are termed *local*

nodes for that processor. For a given processor, the set of all nodes that are either local or shared is termed the processor's *active node list*.

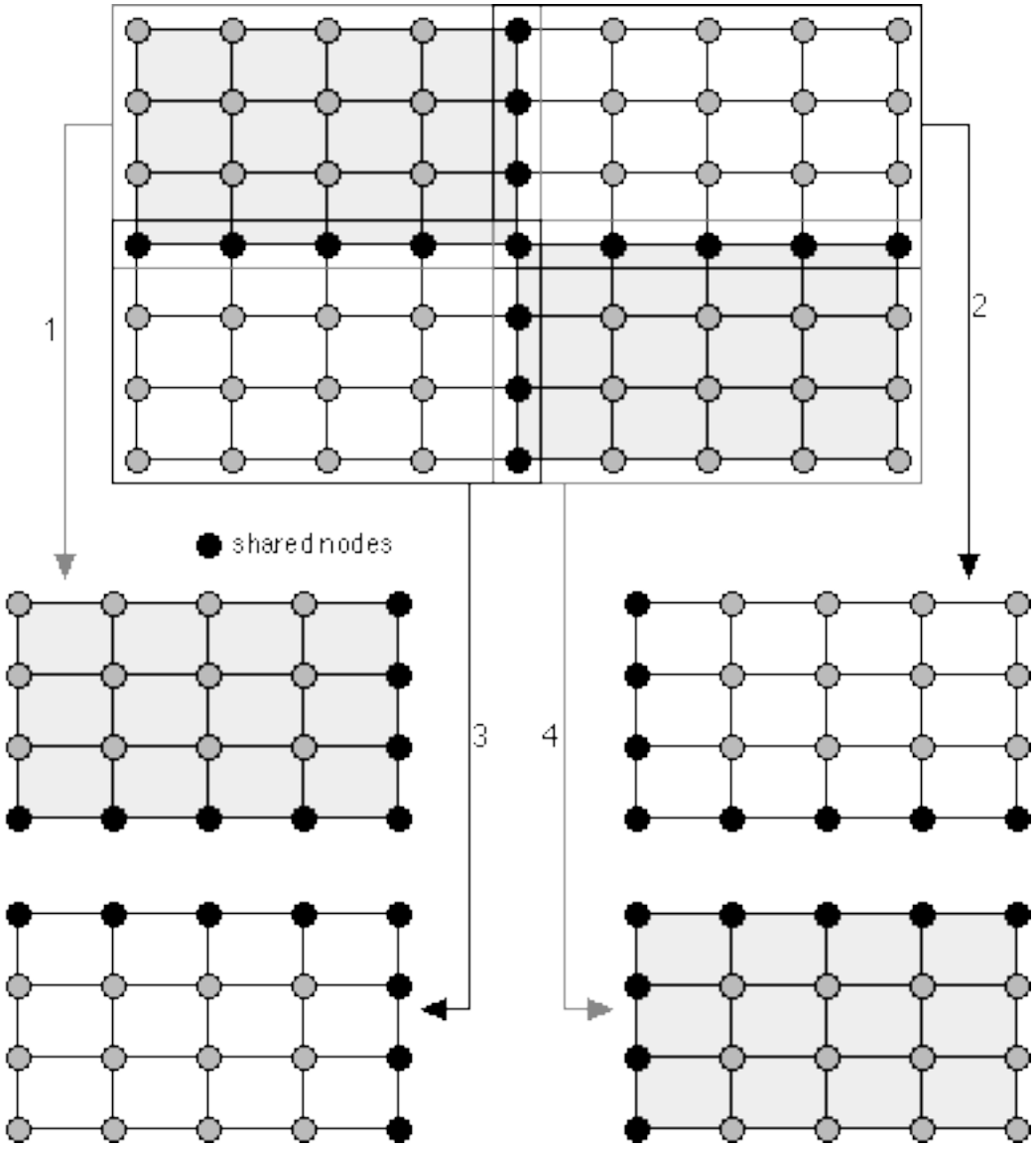


Figure: decomposition of a domain into four processor subdomains

Active nodes

The set of active nodes can be defined as the list of all nodes associated with elements located on a given processor. This set can be constructed by scanning all the elements on a processor while accumulating the nodes associated with each element into a list of distinct node ID numbers. This set is henceforth termed *the processor active node list* (or in more common shorthand, *the active node list*).

In addition, this notion of active nodes can be extended to consideration of an active node list associated with aggregrations besides the obvious one of "all nodes associated with elements lying on a given processor". In particular (see the fundamental definition of *block* given below), one very important subset of the processor active node list is the set of all nodes on a given processor associated with a given block of elements: this particular active node list plays an important role in returning the computed solution to the parent finite-element program.

Shared Nodes

The set of shared nodes can be precisely defined as the list of all active nodes that cannot be associated with a unique processor. Shared nodes are thus a subset of the set of active nodes, and can be equivalently characterized as nodes that can be found in the active node list for more than one processor. From a practical standpoint, shared nodes are found on the boundaries of elements that lie along the edges/faces of the individual subdomains resulting from the domain decomposition process (as was shown in the preceding diagram for domain decomposition).

Thus, the domain decomposition carried out by the parent finite-element code already identifies shared nodes, hence this data is readily available for passing to the equation solver module. The alternative of determining the list of shared nodes without any help from the parent finite-element code represents an extremely complicated and expensive computational process, and avoiding this expense is the main reason why the parent finite-element program is given responsibility for identifying all shared nodes.

External Nodes

External nodes are nodes that are needed on a given processor for subsequent calculations, but are not located on that processor, either as local or as shared nodes. The most common case of an external node occurs when a slidesurface constraint crosses a processor boundary, so that (as a concrete example) a slave node located on another processor is coupled to one or more master nodes present in a processor's active node list. In this case, all of the relevant interprocessor communications pathways cannot be resolved simply by scanning the lists of active and shared nodes. In this setting, the parent finite-element code is responsible for advising the solver of the location of the external node, as this information is also generally available to the calling finite-element program (where it is used in determining the nodal weights that define the constraint relation's precise mathematical representation).

This case of external node characterization is diagrammed in the figure below. Note how the slave node is not present in the active node list corresponding to the processor where the master nodes are found. Note that in this case, the slave node *s* from processor 1 is an external node for processor 2, and the master nodes m_1 and m_2 from processor 2 are external nodes for processor 1. In both cases, the calling finite-element program bears the responsibility for advising the solver of each instance of external nodes.

The parent finite-element program is responsible for informing both processors of the send/receive characteristics of all external nodes. In this example, the existence of the constraint involving slave node *s* (as external to processor 2) and master nodes m_1 and m_2 (external to processor 1) must be declared by the calling finite-element program. This declaration process involves articulation of both the send and receive characteristics implied by the constraint relation. In particular:

- processor 1 needs to **send** data involving node *s* to processor 2, while **receiving** data defined at nodes m_1 and m_2
- processor 2 needs to **send** data involving nodes m_1 and m_2 to processor 1, while **receiving** data defined at node *s*.

The finite-element interface provides appropriate initialization methods so that external nodes can be identified and categorized for both send and receive communications. With these initialization routines performed by the calling finite-element program, the communications requirements for handling external nodes in a distributed-memory setting can readily be managed.

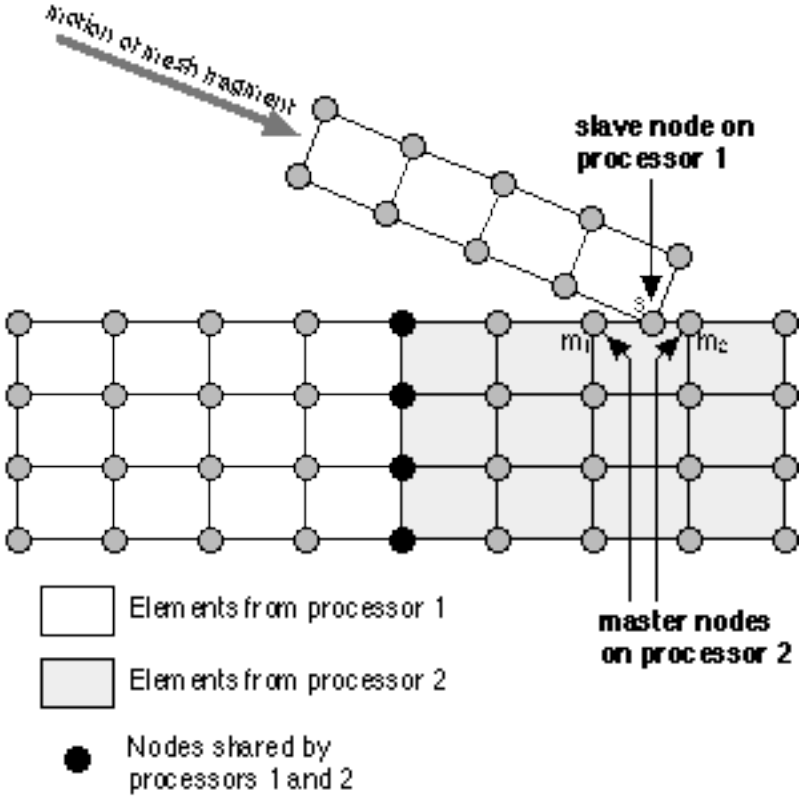


Figure: External (slave) node arising from slidesurface constraint

Block

Nodes, elements and constraints thus naturally arise in finite-element approximation, and elements are commonly derived from a generic set of elemental approximations called an *element library*. Thus there is generally some standardization of types of elements (e.g., the number of nodes, the number of solution unknowns to be determined at each node) present within a typical finite-element code. This standardization leads to the concept of a *block* of elements.

Block Definition

A block is a collection of elements lying on a single processor and satisfying two specific criteria:

- all elements have the same number of associated nodes, and
- all associated nodes have the same pattern of solution unknowns.

The former criterion means that elements belonging to a given block have standardized data structures that can readily be allocated: one obvious example is that the element topology (i.e., the list of nodes associated with a given element) for all elements in a given block can be stored in a typical two-dimensional matrix, with each row representing the topology for a given element. In this case, all row lengths are the same, hence the resulting data structure is readily stored in a Fortran-style two-dimensional array.

Block Nodal Solution Cardinality

The second criteria is more subtle, as care must be taken to distinguish between blocks where each element's associated nodes have the same solution cardinality (i.e., the same number of nodal unknowns is defined at each node), and blocks where each element has a variable solution cardinality. The former case is shown in the figure below, where each node is identical to all the others associated with a given element (e.g., in a thermal problem, each node has one unknown, in a 2D compressible solid mechanics problem, each node has two unknowns, etc.).

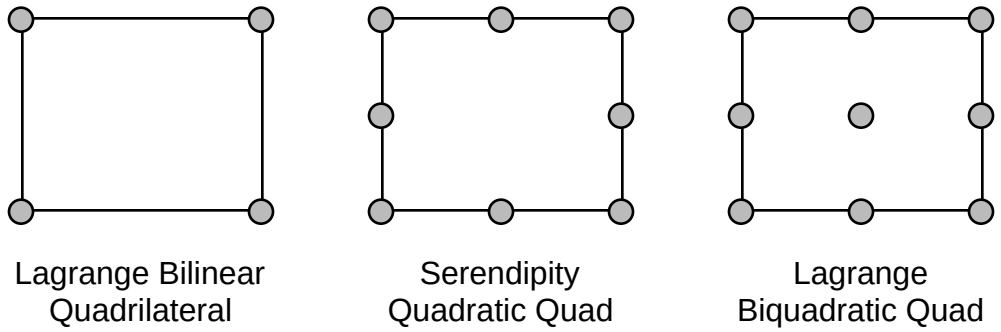


Figure: Generic Elements from Blocks with Constant Nodal Solution Cardinality

When solving multi-physics problems (such as a couple solid-thermal analysis), the different solution fields are not necessarily determined on identical sets of nodes. In fact, in some coupled problems, solvability conditions required for convergence of the FE analysis often *require* that different orders of approximation be utilized for the various solution fields present in the computational analysis. In this case, the notion of a constant solution cardinality for each node in a given element must be jettisoned, to be replaced by a *list* of solution cardinalities, each associated in order of the local numbering of the element topology.

Some examples of such varying solution cardinality elements are shown below: the elements on the right and left are commonly used in analyses of incompressible materials, where the higher-order interpolant (the bilinear and biquadratic node sets, respectively) are used to interpolate the displacement field, and the lower-order interpolants (the constant and linear fields, respectively) are used to estimate the piecewise-discontinuous pressure field that often accompanies incompressible analyses. The middle element is commonly used in porous-media-flow problems and thermal/solid analyses, where both solution fields must be interpolated in a continuous manner (though in this case, with different-order interpolants).

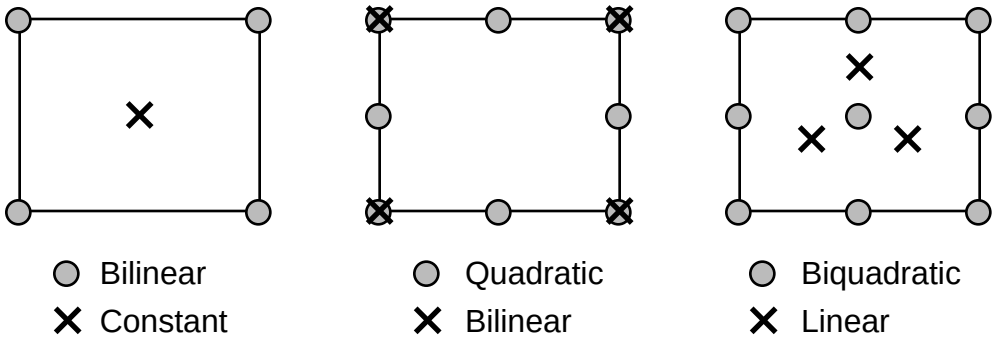


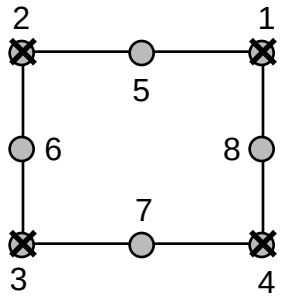
Figure: Generic Elements from Blocks with Variable Nodal Solution Cardinality

Once the notion of a nodal solution cardinality pattern is admitted, then care must be taken to utilize a consistent local and global numbering scheme so that the correct local nodes (i.e., nodes associated with a given element) can be identified with their respective nodal solution parameters as the finite-element equations are assembled, solved, and the solution components returned to the parent finite-element analysis.

This numbering consistency issue is easier to grasp by using a simple example: first, consider the local numbering scheme imposed on the mixed-order interpolant defined over the element pictured below, where there are two coupled fields:

- a vector displacement field utilizing a quadratic eight-node serendipity element with two unknowns per node, and
- a scalar potential field (e.g., temperature) defined on a superimposed bilinear Lagrange element with one unknown per node.

The table given in the figure gives the solution cardinality in terms of the local element numbering: this latter numbering scheme begins at the upper-right-hand corner of the element, proceeds around the corner nodes (where both displacement and potential are interpolated) in the usual positive rotational sense, and then winds around the element boundary a second time to catalog the mid-side nodes, where only the two displacement vector components are interpolated.



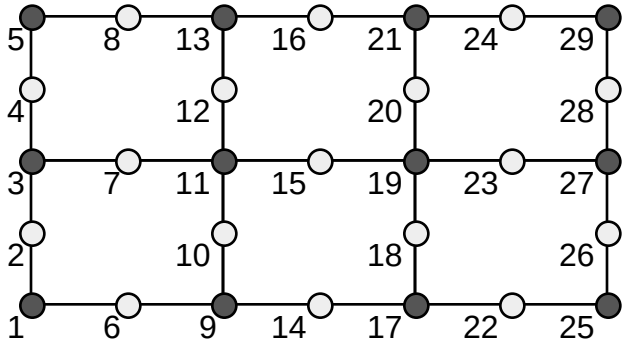
FE approximation with two unknowns per node

FE approximation with one unknown per node

Node	1	2	3	4	5	6	7	8
Cardinality	3	3	3	3	2	2	2	2

Figure: Local Node Numbering for Element with Variable Solution Cardinality

The reader should insure that the table of "nodes and cardinality as a function of local node number" is thoroughly understood before proceeding on to the next figure. In this figure below, a coarse two-dimensional mesh is shown with the usual numbering arrangement obtained by proceeding consecutively across the least mesh dimension (in this case, five node in the vertical direction). As in the last figure, corner nodes possess three unknowns per node, with mid-side nodes having two unknowns per node. Thus the darker circles represent nodes with one additional solution component to be interpolated at each node.



3 unknowns per node

2 unknowns per node

Figure: Sample Mesh with Variable Solution Cardinality

The table of element topology is given in the figure below. Note that the first four columns represent the topology induced by the counter-clockwise numbering of the corner nodes for each element (e.g., for the first element in the lower-left corner, these nodes are 1, 9, 11, and 3), and the last four columns of the topology array represent the counter-clockwise numbering of the mid-side nodes (e.g., for the first element, nodes 6, 10, 7, and 2). Such consistency in local nodal numbering for every element

in a given block is absolutely necessary in order that the processes of stiffness matrix allocation, element matrix loading, and solution return be properly implemented so that the finite-element analysis and the solver module agree on the view of what physical problem is being solved.

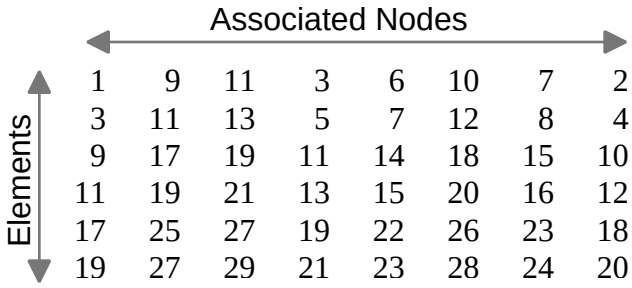


Figure: Permissible Element Topology for Mesh with Variable Solution Cardinality

Block Structure in the Finite-Element Mesh

In practice, blocks of elements are collections derived from the same generic type in an element library. For example, a group of four-node 2D elements with two displacement components at each node can be considered to form a block, while a group of four-node 2D elements, some with (vector) displacement interpolants and others with a (scalar) temperature interpolant, cannot be considered as a block.

Another case of block structure can be developed by considering the transition of a higher-order interpolant into a lower-order approximation. In the figure below, there are three blocks:

- the set of eight-node serendipity quadratic quadrilaterals on the left,
- the intermediate region of five-node transition elements, and
- the set of four-node bilinear quadrilaterals on the right.

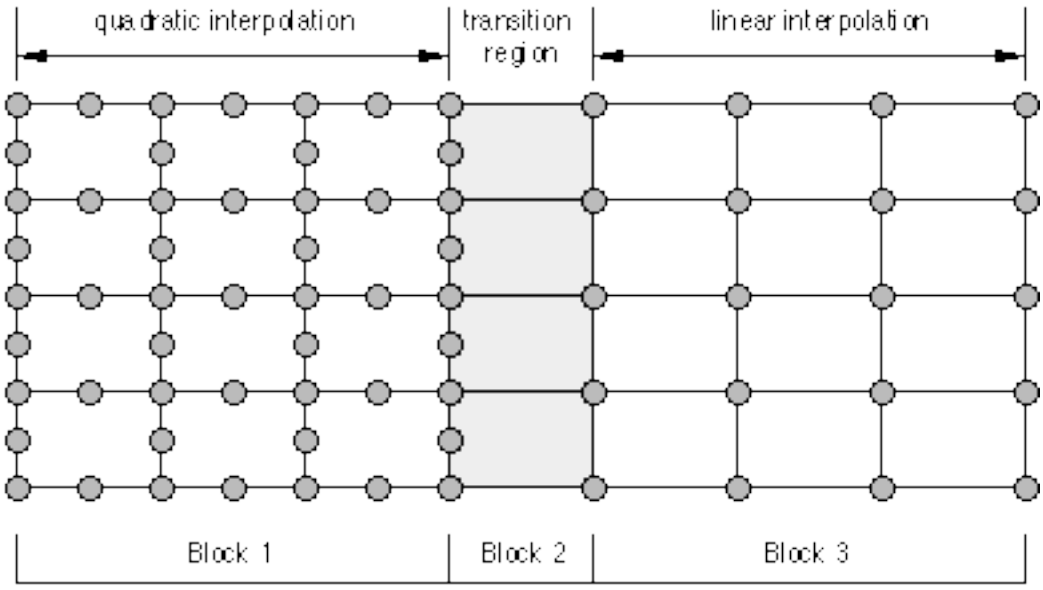


Figure: various blocks present in a transition region

From the standpoint of the finite-element developer, blocks may be further subdivided into smaller classes based on physical constructs, such as material type or number of constitutive parameters. It is important to note that these physical distinctions are not imposed by the solution module, however, so it is not *necessary* to propagate such distinctions into the solution services tasks. The only issues involved in the definition of a block arise from purely elemental topological and solution cardinality concerns – if it simplifies the overall analysis process by grouping disparate physical materials into a block of elements, then the FE developer should do so. If such aggregation complicates life for the developer, then physical distinctions should be propagated between groups of elements that could be otherwise be encapsulated into a single block.

As an example, consider the mesh shown below, which is composed of two distinct materials, but one underlying type of elemental approximation. The region can be decomposed into two blocks, mirroring the material discontinuity at the center of the mesh, or joined into one block for the process of passing data to the solver. The particular choice is best left to the FE developer, to be made on the grounds of whatever is simplest to implement.

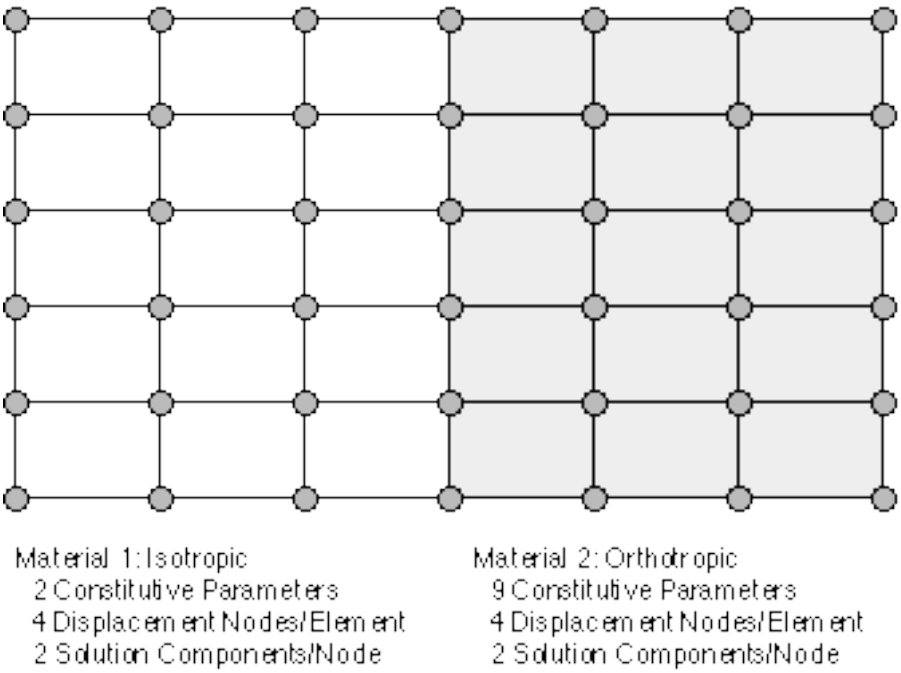


Figure: Problem with two natural blocking strategies

Block Structure Pathologies for Further Study

In many multi-physics problems, different regions of the solution domain may utilize different numbers of solution components. For example, a solid mechanics problem over a geometric domain may be overlaid by a thermal analysis that is coupled to the solid analysis over a smaller subdomain where the temperature response is of particular interest. Because the two overlapping regions possess different solution cardinalities, they must be mapped to two (or more) separate blocks of elements.

Once multiple blocks possessing different solution cardinalities are permitted in a finite-element mesh, the possibility arises that the nodes along the interfaces between blocks may inherit different views as to the number of solution parameters to be interpolated along this block-contact interface. As a concrete example, consider the figure below, which represents a junction of two blocks:

- the block on the left, where displacement is interpolated over nine-node Lagrange biquadratic elements, and
- the block on the right, where displacement is interpolated over similar nine-node Lagrange elements, but a scalar potential field is overlaid on the displacement mesh, using four-node Lagrange bilinear elements.

From the standpoint of determining the displacement field (i.e., the portion of the solution that is common to both blocks), the two mesh components are

mathematically conforming, and the element matrix contributions to the displacement strain energy functional can be assembled readily by the solver. But the nodes present in the right-hand block for approximating the potential field have no analogous component in the left-hand block, and some means for rectifying this inconsistency must be found.

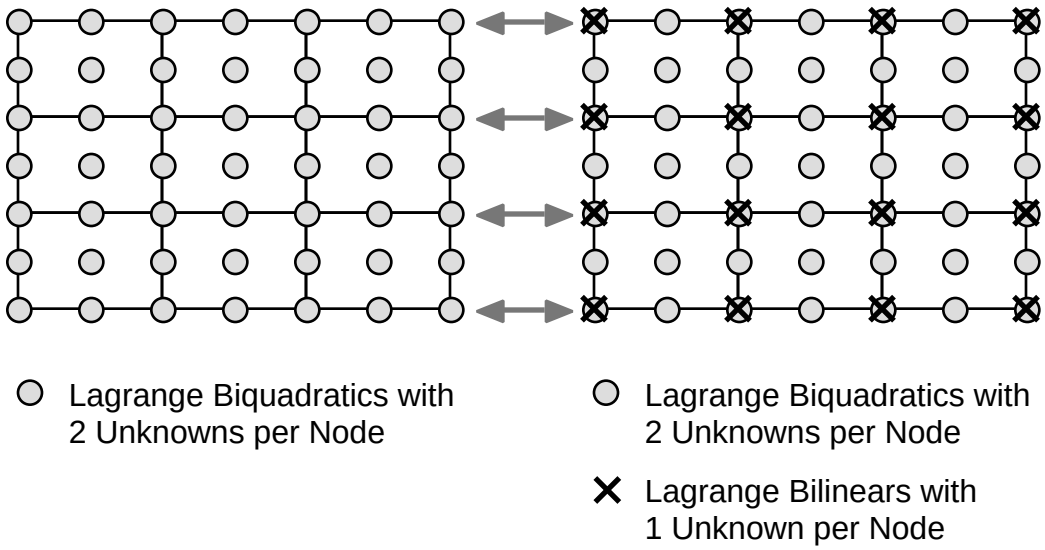


Figure: Solution Cardinality Disagreement at a Block Interface

With the current FE-LA solver interface, this multiple-block structure can be modeled by using distinct nodes for the boundaries of the two blocks, so that the nodes at either ends of the arrows in the figure above can be tied together (for their displacement components, at least) using Lagrange multiplier or Penalty constraint conditions. The thermal problem imposed on the right-hand block must have appropriate boundary-condition data applied to the nodes along the block interface, but this task is left to the finite-element modeler. Thus the current FE-LA solver specification *permits* this multiple-local-field modeling scheme, but it does not explicitly *encourage* it in practice.

If such locally-defined variable-solution-cardinality meshes are to be utilized most effectively using the FE-LA specification (and if these problems are common enough to warrant such a generalization), then some means to permit joining such disparate blocks should be found without requiring the analyst to tie the various components together with constraint conditions. Currently, the plan is to identify such regions where different blocks impose mutually exclusive definitions of nodal solution cardinality on the nodes lying along the block-contact boundary, and then look for nodally-defined data (such as boundary-condition specifications) during the initialization and load steps to determine whether these mutually exclusive cardinality definitions are to be considered as errors in data-passing (and hence signaled as exceptions) or as artifacts of the physical modeling process (and hence dealt with appropriately in the construction and solution of the system of equations).

There are good reasons for extending the interface specification to encourage such block-contact problems. Among the most important of these is the fact that the problem mentioned above results in a *positive-definite* matrix when block-interface conditions can be dispensed with by reusing the same node numbering along the block-contact region, but becomes *indefinite* when a Lagrange multiplier constraint set is used to join the blocks using distinct node numbering along the block contact boundary. Since indefinite problems can be much more difficult to solve via iterative linear algebraic schemes (such as scalable Krylov solvers), imposing such constraints as a modeling technique may have unpleasant ramifications for the linear algebra modules called by the finite-element analysis. Therefore, a means to join blocks without explicit constraint equations may be an excellent idea for study in future versions of the FE-LA solver interface specification.

Element Sets

Blocks represent large collections of elements located on a given processor, and it is common for one block to span the entire processor domain. From the standpoint of passing data to the solver in a parallel setting, a more lightweight entity than a block must be developed, and that construct is termed an *element set*.

Element Set Definition

Element sets are subgroups of elements that comprise element blocks. An element set can range in size from a single element up to the list of all the elements on a given processor. Because element sets are contained within blocks, they also satisfsy the two defining block criteria, namely:

- all elements have the same number of associated nodes, and
- all associated nodes has the same pattern of solution unknowns.

Element sets are simply the means to the end of passing the data that defines a block of elements that is located on a given processor. In practice, two important specific examples should be mentioned:

- the case where each element set has exactly one member, which corresponds to passing element-based data to the solver on an element-by-element basis, and
- the case where elements are grouped to fit within a high-performance memory hierarchy (i.e., a cache of fast RAM with a well-defined size).

This latter case of element set is commonly termed a *workset*. Because element sets are used to decompose portions of blocks located on a given processor, the following criteria must also be satisfied:

- all elements in an element set are local to the same processor, and
- all nodes associated with the collection of elements in an element set are either local to this processor or shared by this processor.

The figure below shows a typical decomposition of a block of elements that spans two processors into four distinct element sets. Note that the number of elements in a set may vary substantially, depending upon the underlying software architecture of the parent finite-element code, or on the particular CPU characteristics (e.g., the amount of cache memory on each processor) of a given computer.

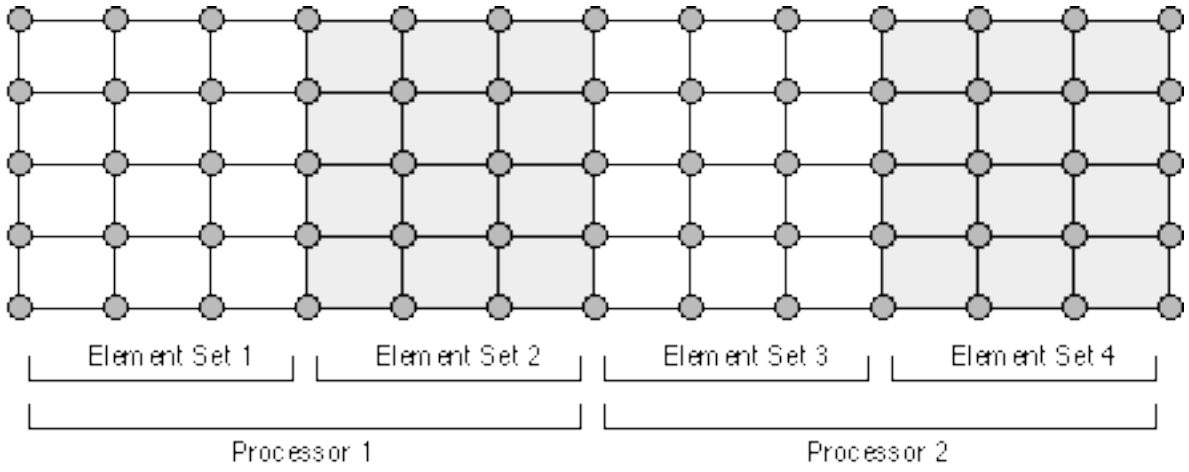


Figure: decomposition of two-processor block into four element sets

The element sets on a given processor satisfy the following criteria:

- the union over all the element sets of all the elements contained in the collection of element sets is the entire list of elements associated with the processor, and
- the intersection over all the element sets of all the elements contained in the collection of element sets is empty.

Furthermore, the union over all element sets *and* over all the processors yields the complete list of all elements in the problem domain, and the intersection over all element sets and all processors is also empty. All of these criteria follow directly from the base construct of domain decomposition.

Element Set Data-Passing Formats

There are many ways to store the element data that forms the bulk of the system of finite-element equations. In particular, many self-adjoint problems result in symmetric stiffness matrices, and this symmetry can be utilized to reduce the memory required to pass element data across the finite-element/solver interface (though it should be noted that symmetry of some or all of the element matrices may not always imply symmetry of the resulting finite-element matrix: asymmetries may result from some specialized types of constraint conditions and from having any of the element matrices be non-symmetric – this latter case commonly arises from material nonlinearities arising when non-associated plasticity models are utilized for constitutive response).

Because it is desirable to achieve the reduced memory footprint that symmetric element stiffness matrices imply, the following set of element formats will be supported by the finite-element/solver interface (note that other formats may be added in the future, depending upon demand):

- dense unsymmetric storage, where the element matrix is stored without any regard for symmetry,
- packed upper-symmetric storage, where the upper triangle of the element stiffness matrix is stored in a row-contiguous manner,
- packed lower-symmetric storage, where the lower-triangle of the element stiffness is stored in a row-contiguous manner, and
- Fortran-style storage, where constant column-length offsets simplify the task (relative to C/C++, at least) of decomposing the element-set-based matrix lists into individual element matrices.

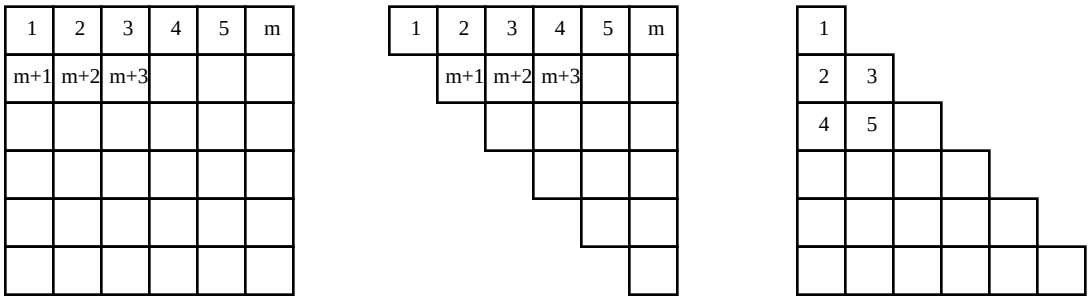


Figure: Base Formats for Element Matrix Storage

Other formats (such as general sparse element matrix storage, for problems with large numbers of solution components at each node, and where the solution coupling is weak or non-existent, resulting in many trapped zeros in the element matrices) will be added if they are technically feasible and commonly used.

Another important issue in terms of element matrix formats is to identify standards for passing element arrays associated with blocks that utilize local elemental unknowns (such as the element-based pressure fields that have heretofore been idealized as being associated with nodes lying within the element, but which can alternatively be viewed as solution unknowns that are defined locally on each element without any idealization using internal nodes). The fundamental idealization used by the current FE-LA solver interface specification is that nodal equations occupy the first elemental equation positions (namely, the lowest-numbered rows and columns), while elemental unknowns occupy the largest-numbered rows and columns.

As a specific example, consider the twelve-node (nine displacement nodes plus three pressure nodes) biquadratic displacement/linear pressure element presented in the "Block Nodal Solution Cardinality" section above. This element can alternatively be idealized as possessing nine nodes, each with two displacement unknowns, and

three scalar elemental unknowns that define a local (element-based, discontinuous across element boundaries) linear pressure field. This element is thus associated with element matrices that have twenty-one rows and columns: eighteen rows arising from the nine sets of two displacement components, and three rows arising from the local linear pressure field.

Using the standardization given above, the element matrices for these mixed elements would take the form given in the diagram below:

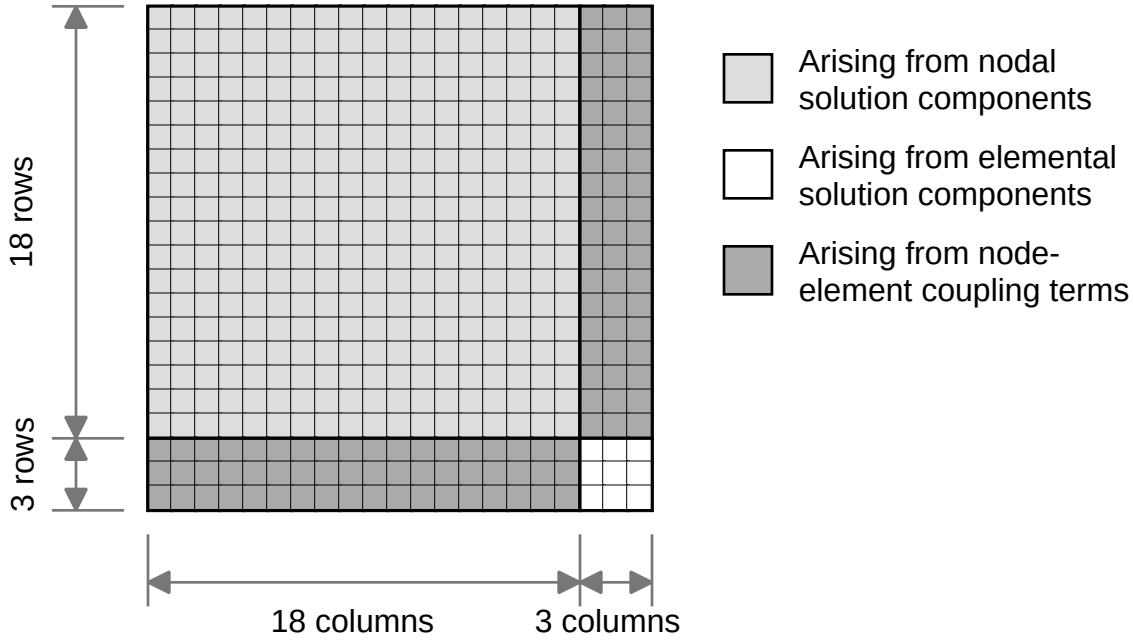


Figure: Sample Storage for Mixed Element with Local Element Unknowns

Node Sets

Just as elements are grouped into aggregrate generic structures termed element sets, nodes can be grouped into similar structures that are called *node sets*. Whenever a group of nodes has some underlying generic similarity, such as a uniform boundary specification, it becomes a natural candidate for encapsulation into a node set. As in the case of element sets, node sets can readily degenerate down to single members, which corresponds to passing nodal information on a node-by-node basis.

The main distinction between an element set and a node set is that while each element on a processor must occur in some particular (and distinct!) element set, few such constraints apply to the union or intersection of the node sets. In fact, the only cases where nodes *must* be passed via node set aggregation occur when nodes possess boundary condition data, or are shared among processors, or are external nodes for use in constraint relations. For all other active nodes, no additional data

needs to be passed to the solver (except, of course, these nodes must occur somewhere in the element connectivity data used to create the active node list).

The assumptions present with regard to node sets on a given processor thus take the following form:

- the union over all the nodal sets of all the nodes contained in the collection of node sets is the list of nodes that either have boundary data defined, or those that involve communications with another processor (including the case of external and shared nodes), and
- the intersection over all the nodal sets of all the nodes contained in the collection of node sets is in general not empty, as it may include nodal data that has some overlap, such as a node shared between processors that also has a boundary condition associated with it.

Boundary Condition Data

A common example of a node set is a group of nodes satisfying a given boundary condition. In general, boundary conditions utilized in finite-element analyses fall into one of three fundamental categories:

- essential boundary conditions, where a particular nodal value of the solution parameter must be enforced,
- natural boundary conditions, where an applied source or load term (in general, the dual of the primary solution unknown) is added to the global load vector, and
- mixed boundary conditions, that involve a linear combination of essential and natural boundary specifications.

If the primary solution unknown (taken to be a scalar: if the solution is vector-valued, this parameter should be taken as one component of the vector nodal solution) is denoted by u, the dual of the solution (force as opposed to displacement, heat source as opposed to temperature, etc.) is denoted by q, and the nodal values of these solution parameters are indicated by a subscript j, then a generic boundary specification can be given by:

$$\alpha_j u_j + \beta_j q_j + \gamma_j = 0$$

where α_j, β_j, and γ_j are specified constants.

The table below specifies various values for the constants required to produce the three fundamental types of boundary conditions.

essential	$\alpha_j \neq 0$	$\beta_j = 0$	γ_j arbitrary
natural	$\alpha_j = 0$	$\beta_j \neq 0$	γ_j arbitrary
mixed	$\alpha_j \neq 0$	$\beta_j \neq 0$	γ_j arbitrary

Appropriate values for these defining constants are passed for each solution component in the node list through the **alphaBCDataTable**, **betaBCDataTable**, and **gammaBCDataTable** arrays.

Constraints

The final example of data to be passed to the solver via the finite-element interface is that of mathematical constraint relations utilized to implement such useful modeling techniques as contact-impact schemes. These constraints can be cast in a generic form involving weighted nodal summations, and two different schemes can be realized for implementing these constraint equations:

- Lagrange multipliers, where the constraint relation is appended to the finite-element system of equations, is enforced exactly, and involves a new solution parameter (i.e., the Lagrange multiplier) that is often of interest to the parent finite-element program as a statical (i.e., the dual of kinematical, hence representing a force-like quantity) parameter that aids in implementation of the constraint (e.g., a force that enforces a specified displacement constraint).
- Penalty formulations, where the underlying energy functional is augmented by a penalty term that varies quadratically in the residual function corresponding to the constraint. Note that penalty formulation constraints are seldom satisfied exactly, but instead are specified to a precision that is approximately related (in an inverse manner) to the size of the penalty number.

If an interface condition is specified for enforcement with a Lagrange multiplier formulation, then a new row needs to be added to the finite-element equation set, and a new equation parameter (namely the Lagrange multiplier used to enforce the constraint) must be appended to the solution vector. The Lagrange multiplier interface condition takes the following form:

$$\sum_{j=1}^{NN} \underline{w}_j^T \underline{u}_j + f = 0$$

where

NN	number of nodes in constraint
$\underline{u}_j$	nodal solution parameters for each node in constraint
$\underline{w}_j$	nodal solution weights for each node in constraint
f	constant term required for constraint satisfaction

In order to use a Lagrange multiplier formulation for this constraint, a term of the following form is added to the finite-element energy functional:

$$\lambda_k \left[\sum_{j=1}^{NN} \underline{w}_j^T \underline{u}_j + f \right]$$

Minimizing (or more precisely, extremizing, as the addition of this constraint may produce an indefinite system, even when the original finite-element analysis arose from a strict minimization principle) the augmented functional results in a set of equations that satisfy the algebraic constraint. It is worthwhile to recall that if any of the nodes that define the constraint are not in the active node list, then they must be classified as external nodes and appropriate initialization data passed so as to resolve interprocessor communications issues.

If a constraint condition is specified for enforcement with a penalty formulation, then (in general) no rows need to be added to the finite-element equation set, and an especially simple penalty constraint can be applied at the assembled matrix level that requires only minor modification of the sparsity structure of the assembled matrix. The interface condition takes the same form as in the Lagrange multiplier case:

$$\sum_{j=1}^{NN} \underline{w}_j^T \underline{u}_j + f = 0$$

but this penalty constraint can be implemented by adding the following energy term to the problem's strain energy functional:

$$\frac{p}{2}\left(\sum_{j=1}^{NN} \underline{w}_j^T \underline{u}_j + f\right)^2$$

Here, the number p is taken to be a large number (a "penalty number") that penalizes the energy functional greatly for any lack of enforcement of the constraint relation. Since the finite-element analysis corresponds to minimization of an appropriate energy functional, if p is taken sufficiently large, then the constraint (or more accurately, a good approximation of it) will automatically be satisfied by the resulting augmented finite-element energy minimization.

As in the Lagrange multiplier case, if any nodes that define the constraint are not active nodes for a given processor, then they must be considered external nodes for that processor, and appropriate external node data must be passed. In addition, the interface specification provides for various means of selecting the penalty parameter.

Other Constraint Considerations

Constraint data can be managed on either a blocked or an individual basis. A block of constraints represents a series of nodal lists where the parameters that define the constraint (i.e., the nodal weights and the number of nodes present in each constraint) do not change over the various rows in the blocked constraint set. If it is desired to hand constraints to the solver one constraint relation at a time, then the notion of a constraint set can readily be degenerated down to a single constraint by setting the number of constraints in a block equal to one.

As a specific example of how a constraint relation can be packaged in a generic form, consider the transition region between quadratic and linear interpolation shown in

the figure below. In this diagram, a block of elements with quadratic variation of displacement (and hence two unknowns per node) is to be tied to a block of similar elements, except this latter block uses a linear finite-element basis, and only four nodes per element.

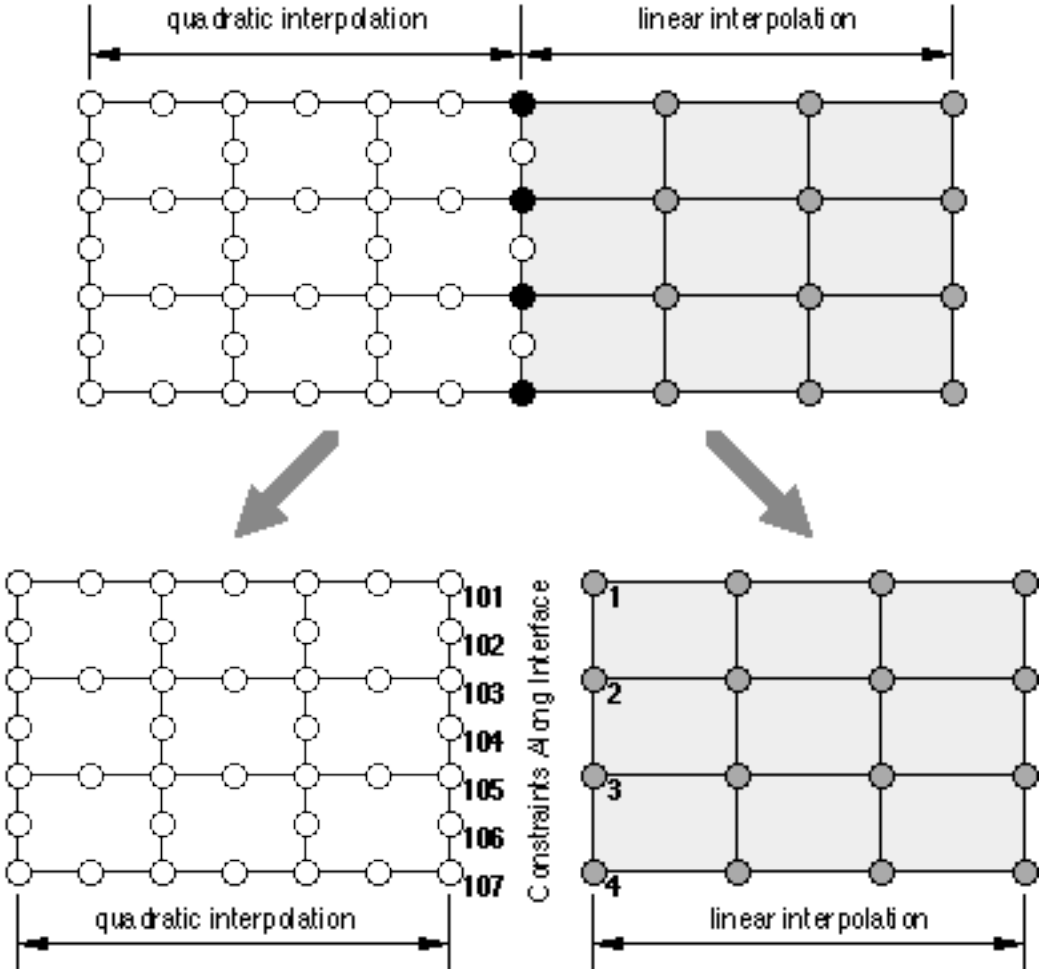


Figure: Constraint Relations at Element Block Interface

In this example, the displacement field at nodes 101, 103, 105, and 107 should be identical to that field defined at nodes 1, 2, 3, and 4. In addition, the displacements at nodes 102, 104, and 106 should be constrained to be the average of the displacements at the pairs of nodes given by (1, 2), (2,3), and (3,4). These results can be written in the following tabular forms, where u and v represent the x and y components of displacement, respectively:

$u_{101} = u_1$	$v_{101} = v_1$
$u_{103} = u_2$	$v_{103} = v_2$
$u_{105} = u_3$	$v_{105} = v_3$
$u_{107} = u_4$	$v_{107} = v_4$

$u_{102} = (u_1 + u_2)/2$	$v_{102} = (v_1 + v_2)/2$
$u_{104} = (u_2 + u_3)/2$	$v_{104} = (v_2 + v_3)/2$
$u_{106} = (u_3 + u_4)/2$	$v_{106} = (v_3 + v_4)/2$

These fourteen individual constraints can be expressed in terms of two pairs of similar constraint sets by loading up the arrays (as defined in the finite-element interface header file) in the following manner:

Constraint on x-component of displacement at the common nodes

ICNodeTable:

101	*1*
103	*2*
105	*3*
107	*4*

ICWeightTable:

1.0	*0.0*
-1.0	*0.0*

ICValueList:

0.0
0.0
0.0
0.0

Note that the same weights are used for each individual constraint, and that the number of rows of the weight array equals the number of columns of the node table (in order to make the product of the nodal unknowns by the weights conformable when there is more than one solution unknown at each node).

For the y-component of displacement, the only change is to the ICWeightTable array, which takes the following form:

0.0	*1.0*
0.0	*-1.0*

To summarize, the rows of the node table represents the individual node IDs associated with each constraint. The columns of the weight table represent the weights applied to each node in a given constraint. The length of the weight table's rows (i.e., the number of columns when each row has the same length) corresponds to the number of solution unknowns defined at each associated node.

Constraint on x-component of displacement at the midside nodes

ICNodeTable:

102	*1*	*2*
104	*2*	*3*
106	*3*	*4*

ICWeightTable:

1.0	*0.0*
-0.5	*0.0*
-0.5	*0.0*

ICValueList:

0.0
0.0
0.0

Note that the same weights are used for each individual constraint, and that the number of rows of the weight array equals the number of columns of the node table (in order to make the product of the nodal unknowns by the weights conformable when there is more than one solution unknown at each node).

For the y-component of displacement at the midside nodes, the only change is to the ICWeightTable array, which takes the following form:

0.0	*1.0*
0.0	*-0.5*
0.0	*-0.5*

Solution Return Issues

The process of advising the algebraic solution module to solve the resulting system of finite-element equations is completely dependent on the particular solver module used (e.g., in this prototypical case, utilizing the ISIS++ parallel solver library). Details on the specifics of communicating with the solver package through the FE-LA solver interface are best found by studying the User's Manual for the particular solver package of interest.

The most relevant generic (i.e., solver-independent) concepts arise in the *solution return* process, when the solver library has constructed a solution to the finite-element equations, and these solution components must be returned to the calling finite-element program. It should be noted that currently, these solution components fall into one of three distinct categories:

- nodal solution parameters that arise from solution unknowns computed for each node in the processor active node list,
- element solution parameters that are associated with blocks possessing local element unknowns (i.e. ones that aren't explicitly associated with a node lying in the element interior, such as the discontinuous pressure-field nodes already presented for use in incompressible solid and fluid mechanics problems), and
- Lagrange multipliers arising from constraint conditions, where these solution unknowns generally represent statical (i.e., force-like) quantities of interest to the parent finite-element program.

Each of these aggregations of solution data is represented by one or more solution return methods provided by the FE-LA solver interface implementation. In particular, the current specification calls for returning nodal solution parameters on the following bases:

- using the list of active nodes associated with a given block to identify and return all the solution parameters associated with a given block,
- using the list of all elements associated with a given block to identify and return the blocked elemental solution unknowns, and
- returning Lagrange multipliers on either a whole-processor basis, or by their aggregation into solution parameters associated with a specific contraint set.

Other solution return functions for specialized applications will be considered in the future according to both demand and the degree of difficulty of implementation. In all cases, the current interface specification is taken to represent a core function set for passing data between the distinct FE-LA modules, and extensions to this base feature set can readily be made to accommodate future demands, as the interface specification is intended to be sufficiently extensible so as to permit growth over its lifespan, as finite-element, solver, and computer technologies evolve with time.

Sample Problems in the Current Specification Distribution

There are several sample data files used to test the current interface implementation, and these sample files can be found using links from the ISIS++ home page on the Sandia/Livermore web site (contact rlclay@ca.sandia.gov for details, as some of these sample problems may have only interim documentation available initially). Pictures and a brief description of the sample problems are given below, in terms of which aspects of the interface implementation each dataset exercises.

Thermal Sample Problem

The thermal sample problem is a simple 2D bar composed of two dissimilar materials that are meshed with non-conforming topology, and then joined using two different constraint sets. This problem exercises several code segments in the prototype

implementation, including handling of essential and natural boundary conditions, and the use of Lagrange multiplier constraint set components.

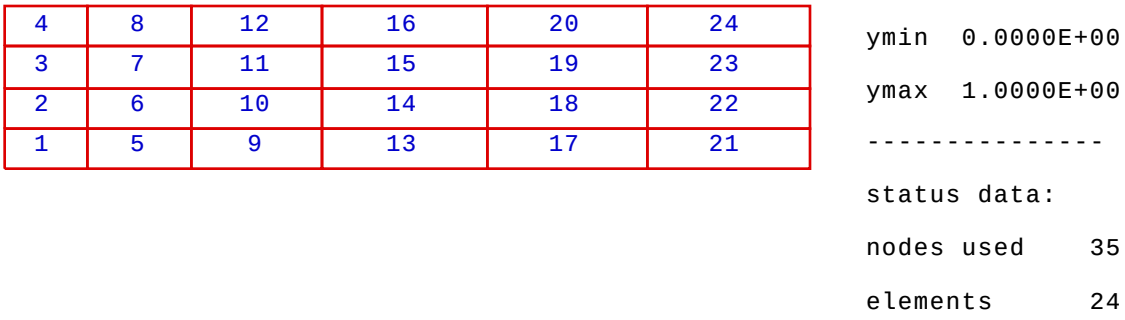


Figure: Thermal Sample Problem Mesh

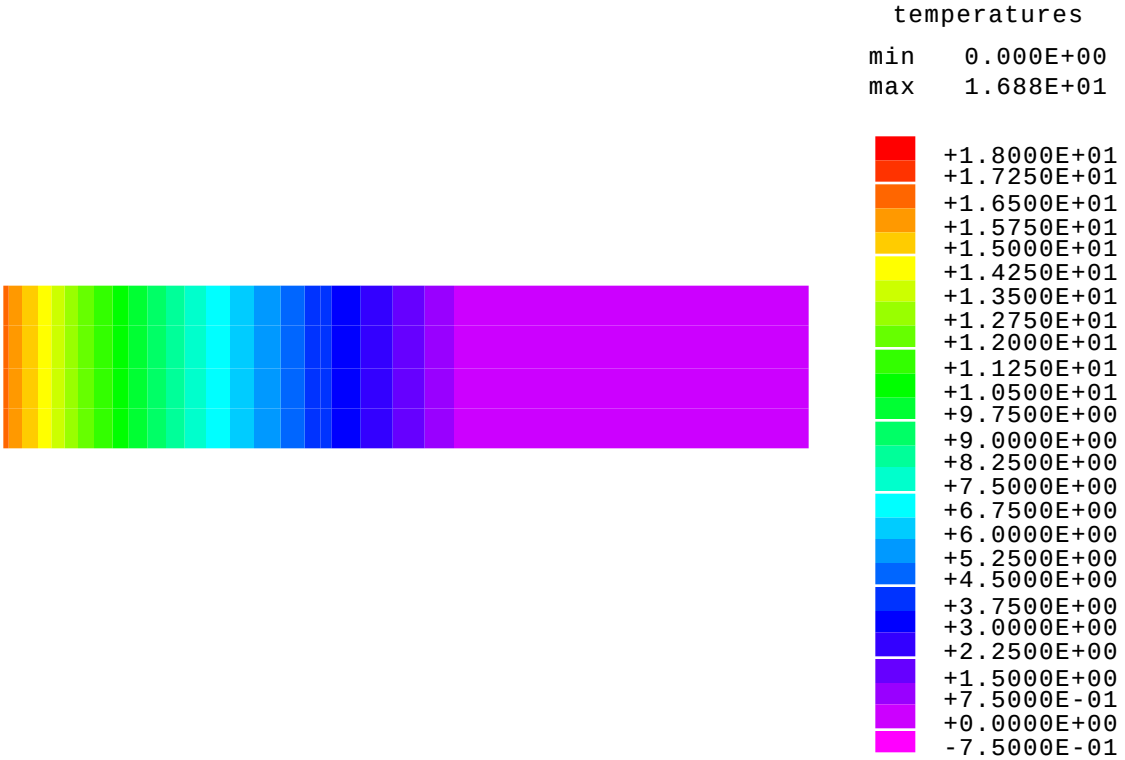


Figure: Thermal Sample Problem Results

Self-Weight Sample Problem

The self-weight sample problem is a ten-element uniform bar composed of 3D brick elements. It represents a bar hanging under its own weight, and tests for vector-valued solution logic, element assembly, and essential boundary-condition handling. In addition, it forms the basis for two more complicated problems that exercise other components of the prototype implementation.

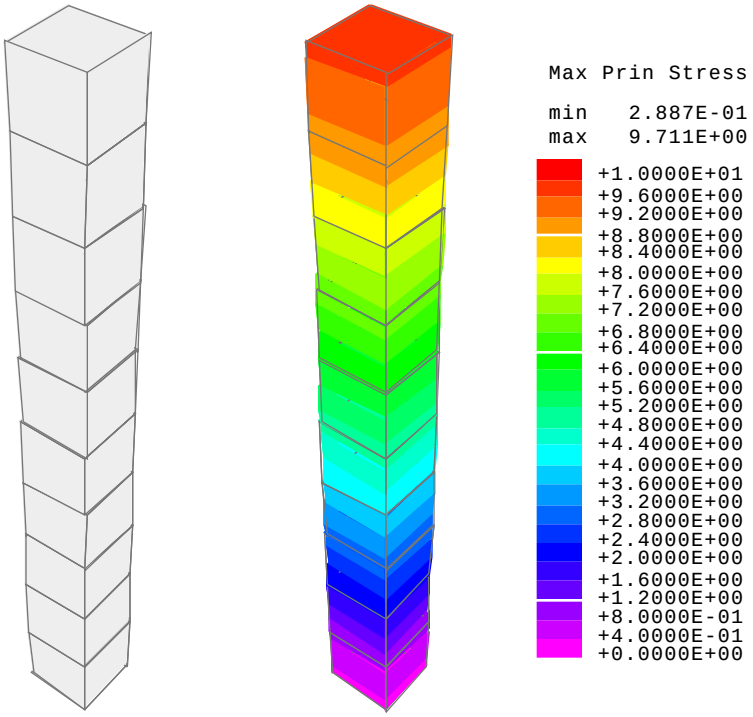


Figure: Self-Weight Sample Problem Mesh and Results

Constrained Self-Weight Sample Problem

The constrained self-weight sample problem represents the same problem used in the self-weight case, but with a support at the bottom of the bar, modeled with a single very stiff (near-rigid) element added to the bottom of the mesh. This additional rigid element causes the bottom of the bar to be supported, and the contact region between the bar and the rigid block is modeled using Lagrange multiplier constraint sets. This sample problem exercises many components of the constraint relation implementation in the prototype, including the solution return functions, where the computed Lagrange multiplier solution parameters represent the reaction forces exerted on the bar by the confining base block.

This problem comes in two flavors: one with contiguous node numbering, and one where the rigid block has nodes numbered in a manner with a gap between the

numbering scheme for the bar and for the block. This latter case provides testing for various globalNode-to-localNode lookup functions in the prototype.

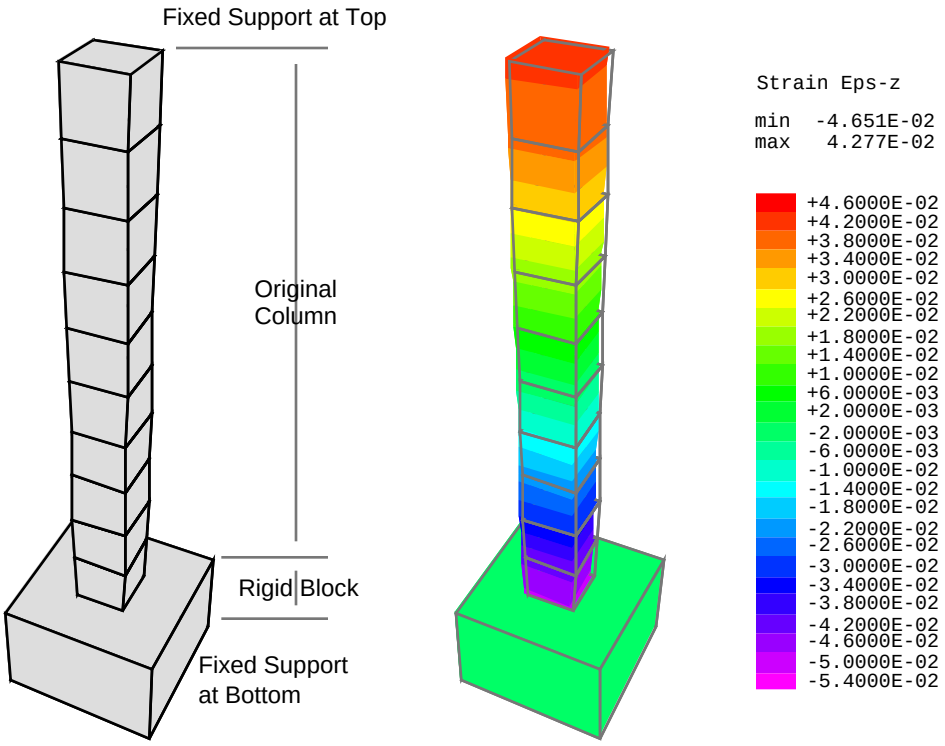


Figure: Constrained Self-Weight Sample Problem Geometry and Results

Three-Dimensional Stress Analysis Problem

This larger problem provides a sufficiently complicated sample problem to permit code profiling of the prototype interface implementation. In addition, it represents half of a larger (symmetric) problem that readily admits testing logic for shared and external nodes in the distributed-memory setting, so it will be used extensively to test scalable versions of the prototype implementation.

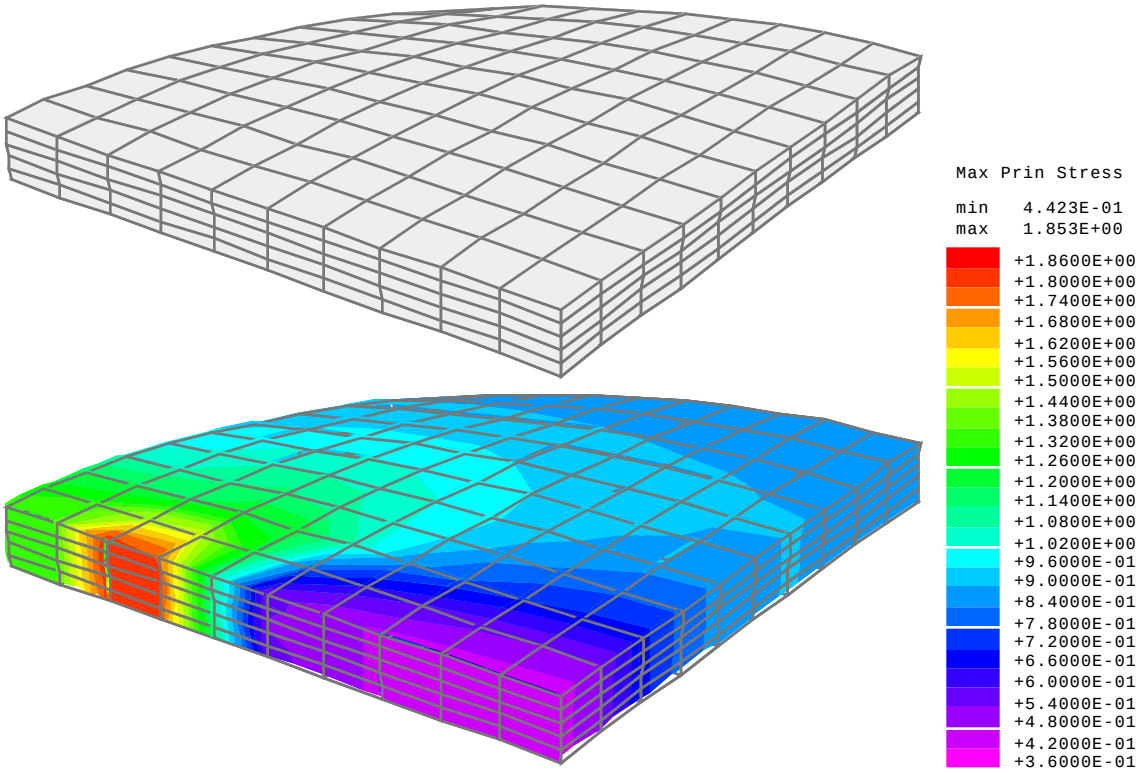


Figure: 3D Stress Analysis Sample Problem Mesh and Results

Knee Joint Stress Analysis Problem

This even larger problem (approximately 12,000 equations) provides another complicated sample problem to permit code profiling of the prototype interface implementation. It tests all components of the prototype implementation except for the constraint relation logic, and it is readily divided (thanks to the uniform topological structure of the mesh) up into various domain decompositions for testing the distributed-memory implementations of the FE-LA solver interface prototype.

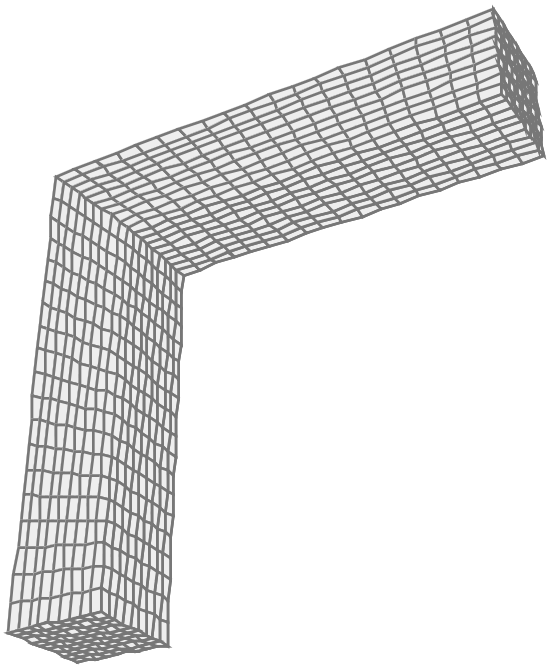


Figure: 3D Knee Joint Stress Analysis Sample Problem Mesh

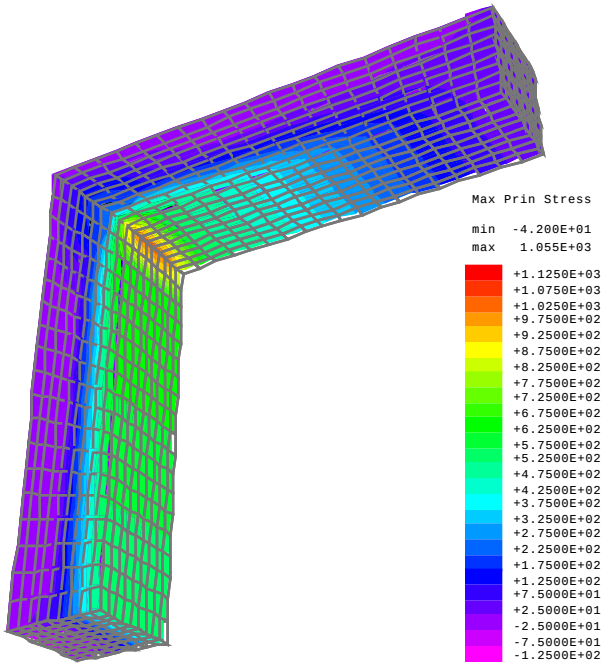


Figure: 3D Knee Joint Stress Analysis Sample Problem Results